UNIVERISTY OF BARI
"ALDO MORO"

Diabetes Detection

IT DEPARTMENT

MASTER DEGREE IN COMPUTER SCIENCE

Case of study in

Machine Learning

Canistro Federico 775605
De Cosmis Ivan 787066

ACCADEMIC YEAR 2022/2023

Contents

Chapter 1

Dataset

This dataset is originally from the National Institute of Diabetes and Digestive and Kidney Diseases. The objective of the dataset is to diagnostically predict whether a patient has diabetes, based on certain diagnostic measurements included in the dataset. Several constraints were placed on the selection of these instances from a larger database. In particular, all patients here are females at least 21 years old of Pima Indian heritage. From the data set in the (.csv) file we can find several variables, some of them are independent (several medical predictor variables) and only one target dependent variable (Outcome).

1.1 Analysis of the dataset

We check the dataset in the *check_dataset* file.

Dataset balance control

It is important to check for balanced data in a dataset because having a balanced dataset for a model would generate higher accuracy models, higher balanced accuracy and balanced detection rate. Hence, it's important to have a balanced dataset for a classification model. This means that each class is given equal priority and can improve the performance of the model. If the data is unbalanced, it can lead to inaccurate results and may require additional steps to balance the data before analysis. So, first of all, to do a good pre-processing analysis we check if the dataset is unbalanced.

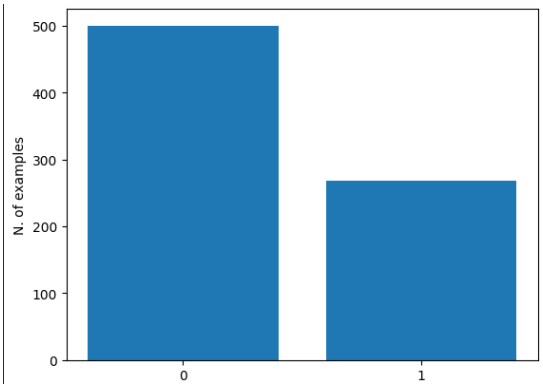


FIGURE 1.1: grapichal rappresentation of distribution of datas between the two classes

As we can see, the dataset is not balanced at all. We have more instances of class 0 than class 1.

Checking for null values

Of course, we check if there are null values in the dataset to understand if there are missing or incomplete data. This can affect the accuracy of the results of a data analysis and it may require additional preprocessing steps to handle these missing values. In this case we found 0 null values, so we don't have to do any operations for solving this issue.

Correlation Matix

A correlation matrix is a table showing correlation coefficients between variables. Each cell in the table shows the correlation between two variables. It is a powerful tool to summarize a dataset and to identify and visualize patterns in the given data. In this dataset the correlation matrix is:

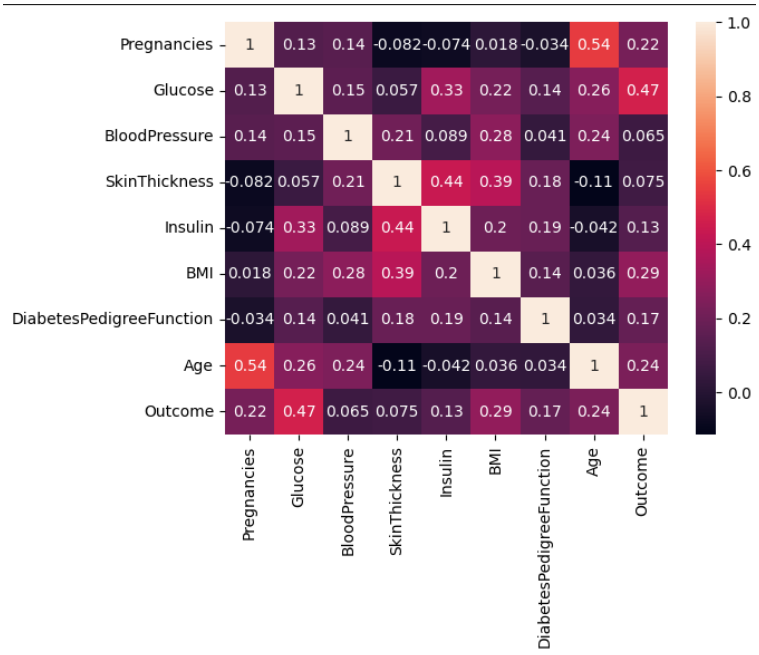


FIGURE 1.2: Correlation Matrix

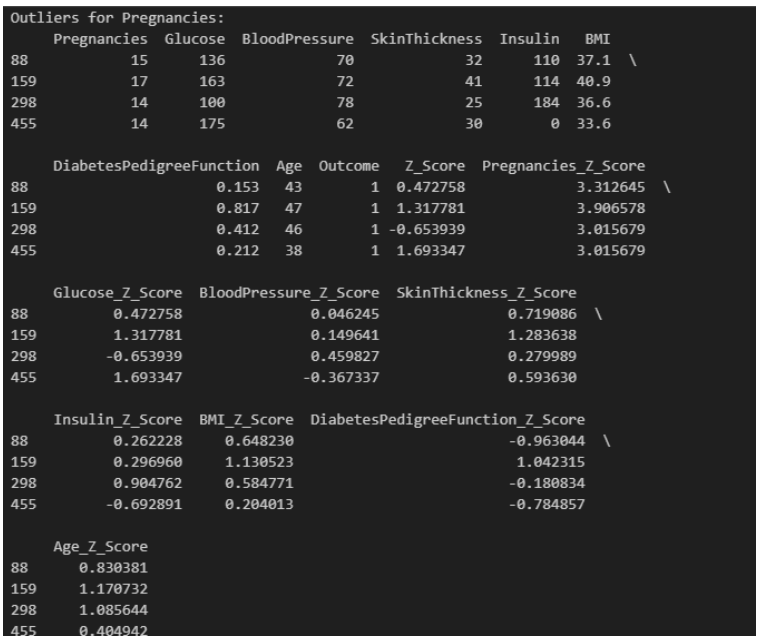
From the matrix, we can see that some features have a stronger correlation with each other than others. For example, *Age* and *Pregnancies* have a relatively strong positive correlation of 0.544341, while *SkinThickness* and *Age* have a relatively weak negative correlation of -0.113970. The *Outcome* variable, which likely represents whether or not an individual has diabetes, has the strongest positive correlation with Glucose at 0.466581. Let's do more analysis.

Z-Score

A Z-score is a measure of how many standard deviations a data point is from the mean of a dataset. It is calculated by subtracting the population mean from an individual raw score and then dividing the difference by the population standard deviation. Z-scores can be positive or negative, with a positive value indicating that the data point is above the mean and a negative value indicating that it is below the mean.

Z-scores are useful for comparing results to a "normal" population and for identifying outliers in a dataset. In the code, Z-scores are calculated for each column in

the dataset (except the outcome column) and then used to identify outliers, which are data points that fall more than 3 standard deviations from the mean 3.



```
Outliers for Pregnancies:
     Pregnancies  Glucose  BloodPressure  SkinThickness  Insulin   BMI
88            15      136             70             32      110  37.1  \
159           17      163             72             41      114  40.9
298           14      100             78             25      184  36.6
455           14      175             62             30        0  33.6

     DiabetesPedigreeFunction  Age  Outcome   Z_Score  Pregnancies_Z_Score
88                      0.153   43        1  0.472758             3.312645  \
159                     0.817   47        1  1.317781             3.906578
298                     0.412   46        1 -0.653939             3.015679
455                     0.212   38        1  1.693347             3.015679

     Glucose_Z_Score  BloodPressure_Z_Score  SkinThickness_Z_Score
88          0.472758               0.046245               0.719086  \
159         1.317781               0.149641               1.283638
298        -0.653939               0.459827               0.279989
455         1.693347              -0.367337               0.593630

     Insulin_Z_Score  BMI_Z_Score  DiabetesPedigreeFunction_Z_Score
88          0.262228     0.648230                         -0.963044  \
159         0.296960     1.130523                          1.042315
298         0.904762     0.584771                         -0.180834
455        -0.692891     0.204013                         -0.784857

     Age_Z_Score
88      0.830381
159     1.170732
298     1.085644
455     0.404942
```

FIGURE 1.3: Part of the output of Z score

We can see from the results that there are some rows in the dataset where the value of the Glucose variable is 0 and not only. This indicates that these values are incorrect, as a value of 0 for blood glucose is not plausible. We can also see that these rows have a very low Z-score (-3.783654) for the Glucose variable. This indicates that these values are far from the mean in terms of standard deviation and therefore we consider them outliers.

T-Student Test

We decided to do a statistical test, by using the test t of student. We choose the t student test because we have a binary classification problem, so this test can be useful in determining whether there is a statistically significant difference between the averages of a continuous variable between two groups (group with diabetes and one without diabetes).
We calculate whether there is a significant difference between the averages of each variable between the two groups:

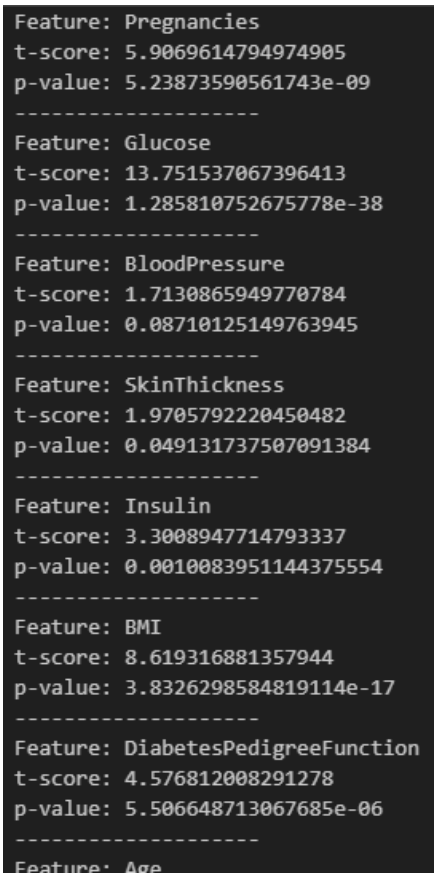


FIGURE 1.4: some outputs of the t-student test

As we can see fig.(1.4), the results show the t-scores and p-values for the Student's t-test for the variables in the dataset. A high t-score and a low p-value indicate that there is a statistically significant difference between the means of the two groups for that variable. For example, for the Glucose variable, the t-score is 13.751537067396413 and the p-value is 1.285810752675778e-38, indicating that there is a very significant difference between the means of the two groups for this variable. Similarly, for the Pregnancies variable, the t-score is 5.9069614794974905 and the p-value is 5.23873590561743e-09, indicating that there is also a significant difference between the means of the two groups for this variable. On the other hand, for the BloodPressure variable, the t-score is 1.7130865949770784 and the p-value is 0.08710125149763945, indicating that there is no statistically significant difference between the means of the two groups for this variable.

Chapter 2

Pre-Processing

Now let's move on to the pre-processing part taking into account what emerged in the dataset analysis part.
As we have seen from the previous analyses, there are no null values in the dataset, but there are incorrect values set to 0 (outliers). So as soon as we read the data, the first thing to do is to replace these missing values. We use a *SimpleImputer* object with the imputation strategy set to "mean" and the missing value set to 0. This object is used to impute missing values in all columns of the dataset except the "Outcome" column. The imputation is done by calling the *fit_transform* method of the imputer object on the relevant columns of the dataset.
Once the missing values are imputed, we separate the input capabilities from the output class by deleting the "Outcome" column from the dataset and storing it in a variable called features. The "Outcome" column is stored in a separate variable called labels.
Then we increase the relevance of significant features that we found in the previous dataset analysis, which are: 'Pregnancies', 'Glucose', 'BMI', 'DiabetesPedigreeFunction', 'Age'.

Normalizing datas

After that, we create a **MinMaxScaler** object and fit it on the input features by calling its fit method. The input features are then transformed using the scaler by calling its transform method and storing the result in a variable called *features_normalized*. Of course we are doing normalization because is an important step in data preprocessing for machine learning because it can help improve the performance of algorithms. Normalization transforms the values of numeric columns in the dataset to a common scale, without distorting differences in the ranges of values. This can improve the accuracy of results and reduce the time complexity of algorithms. Additionally, normalization can help prevent overfitting and improve the training stability of the model.

Splitting datas

The data are then split into training, validation and test sets using scikit-learn's *train_test_split* function. The test set size is set to 0.3 (30%). The training set is the remaining 70% that is further divided into a training set and a validation set with a validation size of 0.2 (30% of the training set).
The validation set is a data set used to evaluate the model's ability to generalize during training. It is often used to optimize the model's hyperparameters. It is called a validation set because it is used to validate the results obtained in the training set. If the performance is poor, we would have to modify the model's hyperparameters

and start over with the train until the result on the validation satisfies us. In this way, we can avoid the phenomenon of overfitting, where the model is able to perfectly predict the data used in training but is unable to generalize on new data.

Class Balance

As we know dataset is unbalanced, class weighting is calculated using the *compute_class_weight* function from scikit-learn with the class weight set to "balanced", classes set to the unique class labels in train_labels, and y set to train_labels. The resulting class weights are stored in a dictionary *called class_weights_dict*.

Saving datas

Then, finally all preprocessed data is saved in **Pre_Processed_Data** folder using the *save* function from numpy.

So we have:

- class_weights.npy
- test_features.npy
- test_labels.npy
- train_features.npy
- train_labels.npy
- val_features.npy
- val_labels.npy

Chapter 3

Classification Algorithms

We have chosen to experiment with 4 classification algorithms for this dataset, and they are:

- K-nearest-neighbor
- Decision Tree
- Support vector machines
- Neural network

K-nearest-neighbor

The k-nearest neighbors algorithm (k-NN) is a type of instance-based learning, or lazy learning, where the function is only approximated locally and all computation is deferred until function evaluation. It is a non-parametric method used for classification and regression.
In k-NN classification, the output is a class membership. An object is classified by a majority vote of its neighbors, with the object being assigned to the class most common among its k nearest neighbors (k is a positive integer, typically small).

Oversampling

For this model we decided to apply SMOTE to balance the data, instead of taking into account the weights of the minority class that we calculated in the preprocessing part.
SMOTE stands for Synthetic Minority Oversampling Technique. It is an oversampling technique to address the problem of imbalanced classes in training data. Instead of simply duplicating examples from the minority class, SMOTE generates new synthetic examples.
SMOTE works by selecting examples that are close in feature space, drawing a line between the examples in feature space and drawing a new sample at a point along that line. Specifically, a random example from the minority class is first chosen. Then k of the nearest neighbors for that example are found (typically k=5). A new synthetic example is then generated as a linear combination of the chosen example and one of its nearest neighbors.
This technique can be very effective in providing additional information to the model to help it learn the decision boundary between classes.
So, SMOTE is applied to the training set to balance the classes. This is done by generating new synthetic instances of the minority class until the number of instances for each class is equal.

Grid search

We use a grid search for finding the best hyperparameters for the model on validation set.
Grid search is a method of selecting the best hyperparameters for a machine learning model. A grid of possible values is defined for each hyperparameter, and a model is trained and evaluated for each combination of hyperparameters in the grid. The combination of hyperparameters that produces the best model performance is selected as the best. In our code, the grid of hyperparameters is defined for the *n_neighbors* parameter of the k-NN classifier and each combination of values in the grid is tested using the **validation set** to evaluate the performance of the model. The best combination of hyperparameters found is then used to train the final model.
This is what happen in our code:

1. A grid of values is specified for the *n_neighbors* hyperparameter of the k-NN classifier.
2. A variable is initialized to keep track of the best hyperparameters and best f1-score found during the grid search.
3. The code loops over each value of n_neighbors in the grid and trains a k-NN model with that value of *n_neighbors* using the balanced training set.
4. Model performance is evaluated on the validation set using f1-score as a metric.
5. If the f1-score of the model on the validation set is better than the current f1-score found, the best hyperparameters and best f1-score are updated with the current values.
6. Finally, a final k-NN model is trained using the best hyperparameters found and the balanced training set and predictions are made on the test set.

We chose to use the f1-score as a metric to evaluate performance on the validation set because the F1-score takes into account both false positives (healthy people classified as sick) and false negatives (sick people classified as healthy) and can provide a more complete view of model performance than perhaps accuracy, which may not be the best choice as a metric for finding the best hyperparameters because it may be affected by class imbalance.

Decision Tree

A decision tree is a type of supervised learning algorithm used for classification and regression tasks. It has a hierarchical, tree-like structure that consists of a root node, branches, internal nodes and leaf nodes.
In the context of classification, a decision tree works by recursively splitting the data into subsets based on the values of the input features. At each internal node of the tree, a test is performed on one of the input features to determine which branch to follow. The test compares the value of the feature to a threshold and sends the data to the left or right child node depending on whether the value is less than or greater than the threshold.
The tree is grown until all the data at a leaf node belongs to the same class or until some stopping criterion is met. Once the tree is grown, it can be used to make predictions on new data by starting at the root node and following the branches down

to a leaf node based on the values of the input features. The class label at the leaf node is then returned as the prediction.

Genetic Alghoritm for finding the bests hyperparameters

A genetic algorithm is an optimization method inspired by natural selection and genetics. The genetic algorithm is used to find the best hyperparameters for the decision tree classifier on the validation set.
The algorithm starts by creating a population of individuals, where each individual represents a set of hyperparameters. The hyperparameters in the code are *max_depth* and *min_samples_split*. The population is then evaluated using the fitness function, which in this case calculates the model's f1-score on the validation set using the current hyperparameters. We use of course f1-score again for selecting the hyperparameters because we are in a binary classification problem.
Next, the algorithm performs a series of genetic operations to create a new generation of individuals. These operations include selection (to choose the most suitable individuals for reproduction), crossover (to exchange genetic information between two individuals) and mutation (to introduce random variations in individuals).
This process is repeated for a predefined number of generations. Eventually, the algorithm returns the best individual found, which represents the set of hyperparameters that maximizes the fitness function.
In our code, the genetic algorithm is implemented using the DEAP library. The genetic operators (evaluation, crossover, mutation and selection) are recorded and the genetic algorithm is executed using the *eaSimple* function of the DEAP library.
The method used in the genetic algorithm is tournament selection. In this method, a random group of individuals is chosen from the population and the individual with the highest fitness value is selected for breeding. This process is repeated several times to choose more individuals for breeding.
In our code, two-point crossover is used, which selects two random cut points and exchanges genetic information between the two cut points to create two new individuals.
Also uniform mutation is used, which randomly changes the value of one or more of the individual's genes within a predefined range.
The tournsize parameter controls the number of individuals that participate in each tournament. A higher value of *tournsize* increases selective pressure, favoring individuals with higher fitness values. Whereas, a lower value of tournsize reduces selective pressure and increases genetic diversity in the population.
The tournament selection method is recorded using the *selTournament* function, and the tournsize parameter is set to 3.
This process is repeated for a predefined number of generations. At the end, the algorithm returns the best individual found, representing the set of hyperparameters that maximizes the fitness function.

Training of the model

After finding the best hyperparameters with the genetic algorithm, we instantiated a decision tree model by passing as features the weighted classes, and the best parameters found with the genetic algorithm, trained the model by means of the *fit* function, and finally used the trained model to make predictions about the test set

Random Forest

After the decision tree, we decided to build a random forest model as well, using the same hyperparameter search approach used in the decision tree, and of course also taking into account the weighted classes.
The Random Forest is an ensemble learning method, which means that it consists of a series of classifiers (in this case, decision trees) and their predictions are aggregated to identify the most prevalent outcome. This can help improve model performance compared to using a single decision tree.

Support vector Machine

We decied to create an SVM classifier with a Radial Basis Function (RBF) kernel.

- SVM is a type of supervised learning algorithm used for classification and regression tasks. It works by finding the hyperplane that best separates the data into different classes.
- The RBF kernel is a popular kernel function used in SVM classification. It maps the input data into a higher dimensional space where it becomes linearly separable. The RBF kernel has two hyperparameters: C and γ. C controls the regularization of the model and γ controls the width of the RBF kernel.

Bayesian optimization

Bayesian optimization is a sequential design strategy for global optimization of black-box functions that does not assume any functional forms. It is usually employed to optimize expensive-to-evaluate functions.
In our code, Bayesian optimization is used to find the best hyperparameters for the SVM classifier with an RBF kernel by minimizing the objective function on the validation set. The hyperparameters C and γ control the regularization and kernel width of the SVM model, respectively.
The *gp_minimize* function from the skopt library is used to perform Bayesian optimization. This function takes as input the objective function to be minimized and the search space for the hyperparameters. In this case, the objective function is defined as 1 minus the f1-score (always because we are in a classification binary problem) of the SVM model, and the **search space** is defined as two real-valued hyperparameters: C and γ.
The *gp_minimize* function uses a Gaussian process to model the objective function and an acquisition function to decide where to sample next. The acquisition function balances exploration (sampling in regions where the objective function is uncertain) and exploitation (sampling in regions where the objective function is expected to be low). The *gp_minimize* function iteratively samples points in the search space, evaluates the objective function at these points, updates the Gaussian process model and acquisition function, and repeats until a stopping criterion is met.
After the optimization is complete, the *gp_minimize* function returns the best hyperparameters found during the optimization. These hyperparameters are then used to create a new instance of an SVM classifier with an RBF kernel.

Training of the model

The best hyperparameters found by the optimization are then used to create a new instance of the SVM model. Of course, we also take class weights into account when instantiating the model (also in the objective function).
Then we train the model with *fit* function and use the trained model for making prediction on the test set.

Neural Network

A neural network is a computational model inspired by the brain that consists of interconnected artificial neurons. It processes input data through layers of neurons, adjusting weights to make predictions or perform tasks.

Data pre-processing

We start this experiment loading all the necessary information needed for training and using the model such as:

- Train features
- Train labels
- Valuation features
- Valuation labels
- Test features
- Test labels
- Weights

In the next step we convert each data to PyTorch tensors, in particular the class weight dictionary is converted into a PyTorch tensor of type float. The reason we convert data to tensors is that the tensors are the fundamental units to represent e manipulate data inside deep learning models. The tensor is similar to a multidimensional array that can contain different data types such as integer or floating point numbers. PyTorch use tensors for several reasons

1. **Efficient computation**: PyTorch is optimized for tensor processing. Tensor operations such as addition, multiplication, and activation functions can be efficiently executed using libraries for linear algebra and parallel computing. Utilizing tensors allows PyTorch to leverage its computational capabilities fully.

2. GPU computation support: PyTorch provides native support for hardware acceleration using graphics processing units (GPUs). Tensors in PyTorch can be moved to a GPU, enabling the utilization of parallel computing power to accelerate model training. Transforming data into tensors is a crucial step to take advantage of PyTorch's GPU acceleration.

3. **Automatic backpropagation**: PyTorch implements the automatic gradient backpropagation algorithm, which is fundamental for training deep learning models. During backpropagation, PyTorch calculates gradients of tensors with respect to the loss function. This process requires data to be represented as tensors to perform the necessary automatic differentiation operations.

4. **Integration with other PyTorch functionalities**: PyTorch offers various useful features for deep learning training, such as data handling, model definition, and parameter optimization. All these functionalities are designed to work with tensors as the fundamental data structure, enabling seamless integration and simpler implementation of deep learning workflows.

We use this tensors to create the Data Loaders that will be used with the model.

Network Architecture

The architecture is based on a 5 layer structure where 4 layers are hidden. Each layer is implemented with **nn.Linear**.
nn.Linear is a module of the PyTorch library that is used to implement a layer inside a neural network. Each layer is fully connected. **nn.Linear** does a linear transformation of input data, multiplying the input tensors by weight matrix and adding a bias term (In our case the transformation depends only on the weights multiplied by the input). To avoid overfitting we use the technique called **Dropout**, easy implemented with the module **nn.Dropout**. This tecnique is based on the idea of deleting random neurons inside the neural network during the train(in out case the chance is 0.5). This process reduce the dependence between neurons allowing the model to be more robust and general.
During the inference or valuation step the dropout is disabled and all the neurons are used to calculate the predicted output.

Loss fuction and the optimizer

We define the loss function as **BCELoss**. BCELoss is a measure of discrepancy between the model's predictions and the corresponding binary target labels. The function averages the cross-entropy loss between the model output probability distribution and the ideal distribution represented by the target labels. We define the optimizer using **torch.optim.RMSprop**. It is an optimizer that adjusts the learning rate based on moving averages of past gradients. After several tests we set the learning rate to 0.0001.

Validate the model

During the train we use a validation function that has the goal to check if the model is improving or not, allowing us to implement a tecnique called "Early stopping". This tecnique stops the model train when the performance on the validation dataset no longer improves. The validation function test the model in input with the validation loader and return the BCE loss calculated in the function. If the difference between the calculated BCE loss and the saved BCE loss is higher than the delta threshold than the model has improved and it will be saved as the best model. When the model does not improved for a number of times equal to the "patience" then the train phase is stopped.

Trainig the model

We train the model with the training set and the epoch number depends on the **Early stopping** tecnique previously described. During each epoch we have:

1. loss.backward(): This step is performed after calculating the loss function between the model's predicted output and the target output. The loss.backward() method automatically computes the gradients of the loss function with respect to the model's weights, using the backpropagation algorithm. Backpropagation calculates the gradient of the loss function with respect to each weight in the model, determining how they affect the model's performance. These gradients will be subsequently used to update the model's weights.

2. optimizer.step(): After computing the gradients during the backpropagation phase, the next step is to update the model's weights. The optimizer, uses the RMSprop optimization algorithm. By calling optimizer.step(), the optimizer updates the model's weights based on the previously computed gradients during backpropagation. This weight update allows the model to move in the direction of the minimum of the loss function, improving the model's performance. The learning rate, specified during the optimizer's creation, determines the size of the weight update steps.

After the train the best model is loaded and used for the evaluation on the test set

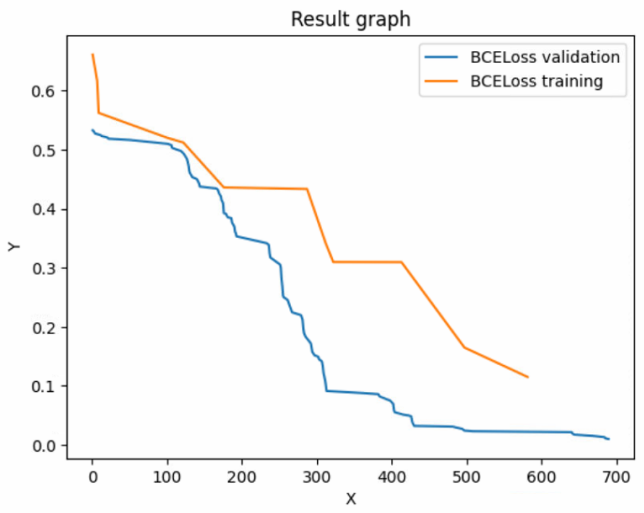


FIGURE 3.1: Resulting graph from the training phase

As we can see, during the test we reach a point where the trend for the training loss is still decreasing while the trend for validation starts to be constant, this is overfitting.

Testing the model

We test the model using the test set and we analyze the output of the model using different metrics. During the test we have used precision to evaluate the model performance.

Chapter 4

Results on test set

As metrics to evaluate the results on the test set, we decided to use the following:

- **Confusion matrix:** A confusion matrix is a table that is often used to describe the performance of a classification model. It shows the number of true positives (TP), false positives (FP), true negatives (TN), and false negatives (FN) for each class.
- **Accuracy:** is a measure of the performance of a classification model. It is defined as the ratio of the number of correct predictions to the total number of predictions. In other words, it measures how many of the cases were correctly classified by the model.
- **Classification report:** The classification report provides a detailed breakdown of the performance of the models for each class. It includes several metrics such as precision, recall, and f1-score. Precision is the ratio of true positives to the sum of true positives and false positives. It measures how many of the positive predictions made by the model are actually positive. Recall is the ratio of true positives to the sum of true positives and false negatives. It measures how many of the actual positive cases were correctly identified by the model. The f1-score is the harmonic mean of precision and recall. It provides a single measure that balances both precision and recall.
- **ROC:** ROC stands for Receiver Operating Characteristic. It is a graphical plot that illustrates the performance of a binary classifier as its discrimination threshold is varied. The ROC curve is created by plotting the true positive rate (TPR) against the false positive rate (FPR) at various threshold settings.
- **AUC:** The AUC can be interpreted as the probability that a randomly chosen positive case will be ranked higher than a randomly chosen negative case by the classifier. A high AUC indicates that the classifier is able to effectively discriminate between positive and negative cases.
- **Confidence interval:** The confidence interval is a statistical tool used to estimate the interval in which the true value of an unknown parameter lies with a certain level of confidence. It can be used to estimate the range in which the true error rate of a model lies on a target data set.
  The confidence interval is especially useful when we have only a small set of test data (which we have), as it provides an estimate of the uncertainty associated with estimating the error rate.
  We set the confidence interval to 95%
- **K-fold cross validation** on the training set: is a technique used to assess the performance of a model and to tune its hyperparameters. It involves dividing

the data into k subsets, and then training and evaluating the model k times, using a different fold as the validation set each time.

Results on Knn

The accuracy on the test set is 0.70, which means that 70% of the test cases were correctly classified.
The confidence interval is $[0.6977, 0.8088]$.
The classification report fig.(4.1) shows that the knn has a precision of 0.87 and a recall of 0.64 for class 0, and a precision of 0.55 and a recall of 0.82 for class 1. The precision for class 0 is 0.87, which means that out of all the cases predicted as class 0, 87% were actually class 0. The precision for class 1 is 0.55, which means that out of all the cases predicted as class 1, 55% were actually class 1.
The recall for class 0 is 0.64, which means that out of all the actual class 0 cases, 64% were correctly predicted as class 0. The recall for class 1 is 0.82, which means that out of all the actual class 1 cases, 82% were correctly predicted as class 1.
The f1-score for class 0 is 0.74 and for class 1 it is0.66.

Classification Report:

	precision	recall	f1-score	support
0	0.87	0.64	0.74	151
1	0.55	0.82	0.66	80
accuracy			0.70	231
macro avg	0.71	0.73	0.70	231
weighted avg	0.76	0.70	0.71	231

FIGURE 4.1: Classification Report

The confusion matrix fig.(4.2) shows that out of 151 actual negative cases (class 0), 96 were correctly predicted as negative (true negatives) and 55 were incorrectly predicted as positive (false positives). Out of 80 actual positive cases (class 1), 66 were correctly predicted as positive (true positives) and 14 were incorrectly predicted as negative (false negatives).

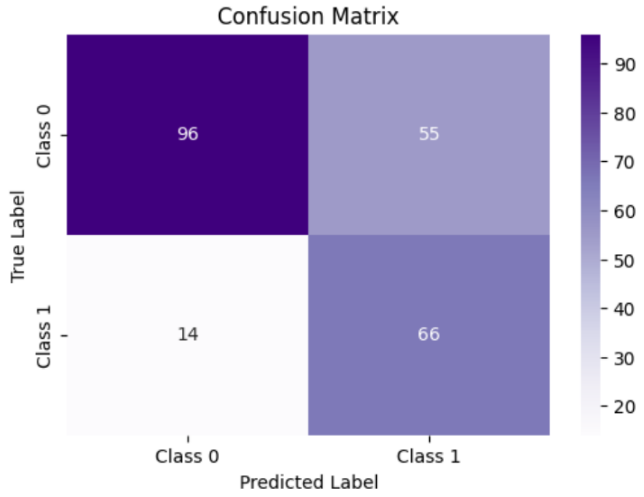


FIGURE 4.2: Confusion Matrix

The area under the ROC fig.(4.3) curve is 0.73, which indicates that the model has good discrimination ability.

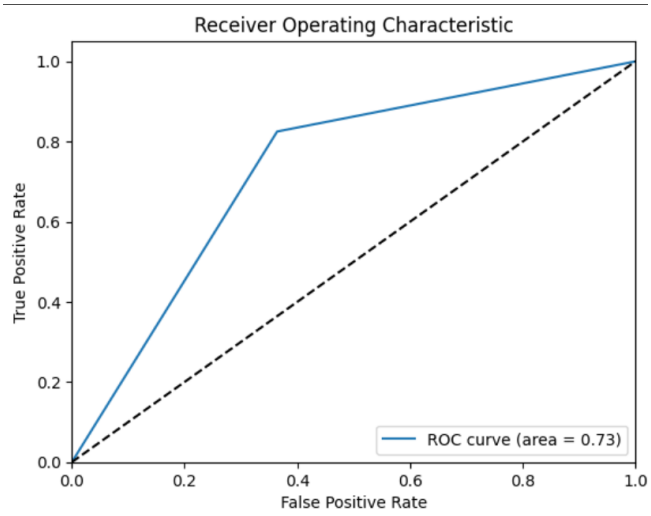


FIGURE 4.3: ROC

Looking at the k-fold cross validation results on the training set, we can see that on average, the knn classifier correctly classified approximately 76.93% of the cases in each fold. The accuracy varied slightly between folds, with a minimum of 73.26% and a maximum of 80.23%.

```
Accuracy scores for each fold: [0.73255814 0.80232558 0.73255814 0.77906977 0.8       ]
Mean accuracy: 0.7693023255813953
```

FIGURE 4.4: Cross validation results

These results suggest that the knn classifier is able to achieve a reasonably good performance on this problem.

Results on Decision Tree and Random Forest

Decision tree results

From the results provided, the decision tree algorithm obtained an accuracy on the test set of 70%, indicating that the model can correctly classify 70% of the instances in the test set.
The confidence interval is $[0.6349, 0.7526]$.
The confusion matrix fig.(4.5) indicates the following results:

- 100 instances were correctly classified as "0" (true negatives).
- 51 instances were misclassified as "1" (false positives).
- 19 instances were misclassified as "0" (false negatives).
- 61 instances were correctly classified as "1" (true positives).

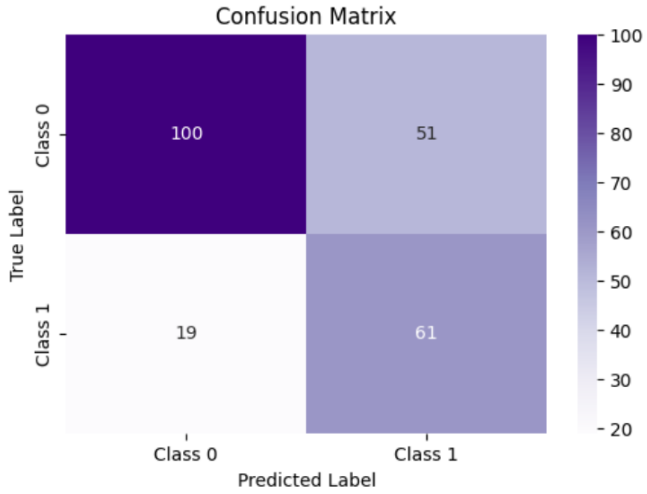


FIGURE 4.5: Confusion Matrix

The precision fig.(4.6) for class "0" is 84%, which means that out of all the instances classified as "0," 84% of them are actually "0." The recall for class "0" is 66%, indicating that 66% of all "0" instances in the test set were correctly identified as such by the model.

For class "1," the precision fig.(4.6) is 54%, indicating that out of all the instances classified as "1," only 54% of them are actually "1." The recall for class "1" is 76%, which means that 76% of all "1" instances in the test set were correctly identified as such by the model.

The f1-score fig.(4.6) for class "0" is 0.74, while for class "1" it is 0.64. The accuracy weighted avg is 0.70, which takes into account the distribution of classes in the test set.

Classification Report:

	precision	recall	f1-score	support
0	0.84	0.66	0.74	151
1	0.54	0.76	0.64	80
accuracy			0.70	231
macro avg	0.69	0.71	0.69	231
weighted avg	0.74	0.70	0.70	231

FIGURE 4.6: Classification Report

The area under the ROC curve (AUC) fig.(4.7) is 0.712, indicating a good ability of the model to correctly classify positive versus negative instances.

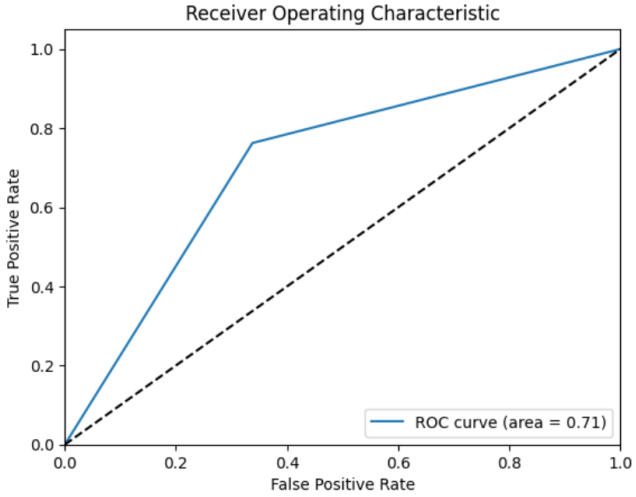


FIGURE 4.7: ROC

We conclude by saying that the results show that the decision tree model has a decent performance in classifying diabetes in the dataset used. Let's see now the results on random forest.

Random forest results

The Random Forest algorithm achieved an accuracy on the test set of 73%, slightly higher than the Decision Tree algorithm.
The confidence interval is $[0.6619, 0.7767]$.
The confusion matrixfig.(4.8) shows the following results:

- 105 instances were correctly classified as "0" (true negatives).
- 46 instances were misclassified as "1" (false positives).
- 16 instances were misclassified as "0" (false negatives).
- 64 instances were correctly classified as "1" (true positives).

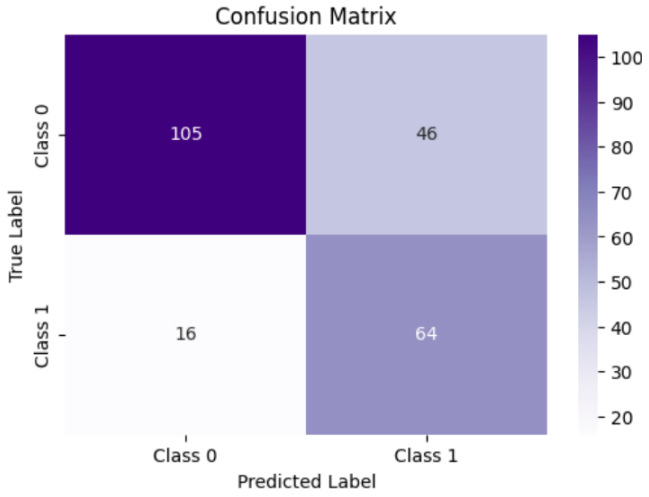


FIGURE 4.8: Confusion Matrix

The precision fig.(4.9) for class "0" is 87%, indicating that out of all the instances classified as "0," 87% of them are actually "0." The recall for class "0" is 70%, indicating that 70% of all "0" instances in the test set were correctly identified as such by the model.
For class "1," the precision is 58%, indicating that out of all the instances classified as "1," only 58% of them are actually "1." The recall for class "1" is 80%, which means that 80% of all "1" instances in the test set were correctly identified as such by the model.
The f1-score for class "0" is 0.77, while for class "1" it is 0.67. The accuracy weighted avg is 0.73, which takes into account the distribution of classes in the test set.

Classification Report:

	precision	recall	f1-score	support
0	0.87	0.70	0.77	151
1	0.58	0.80	0.67	80
accuracy			0.73	231
macro avg	0.72	0.75	0.72	231
weighted avg	0.77	0.73	0.74	231

FIGURE 4.9: Classification report

The area under the ROC curve (AUC) fig.(4.10) is 0.747, indicating a good ability of the model to correctly classify positive versus negative instances.

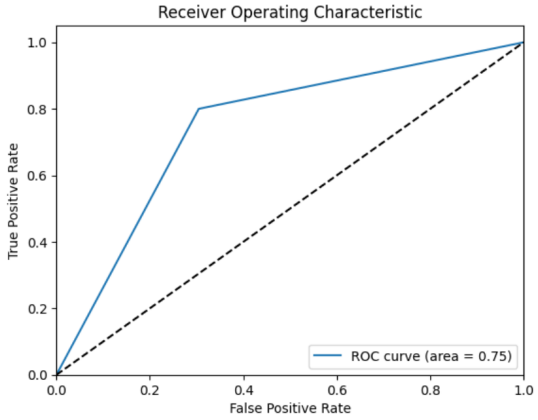


FIGURE 4.10: ROC

Therefore, we can confirm that the Random Forest performed better than the Decision Tree.

Cross validation results

- Decision Tree:
  - Accuracy scores for each fold: [0.75581395, 0.6744186, 0.72093023, 0.69767442, 0.65882353]
  - Mean accuracy: 0.701532147742818
- Random Forest:

- Accuracy scores for each fold: [0.6627907, 0.70930233, 0.72093023, 0.70930233, 0.74117647]
- Mean accuracy: 0.7087004103967168

Considering the average accuracy, we observe that Random Forest has a slightly higher average accuracy than Decision Tree (0.7087 compared to 0.7015). This suggests that the Random Forest may have a greater ability to generalize and fit the data. In fact, the calculation of the metrics confirms that Random Forest is slightly better than Decision Tree in this context.

Results on SVM

As for the SVM model, we obtained the following results:
Accuracy on the test set is 0.71, indicating that the model can correctly classify 71% of the instances in the test set.
The confidence interval is $[0.6574, 0.7727]$.
The confusion matrix fig.(4.11) shows the following results:

- 100 instances were correctly classified as "0" (true negatives).
- 43 instances were misclassified as "1" (false positives).
- 25 instances were misclassified as "0" (false negatives).
- 55 instances were correctly classified as "1" (true positives).

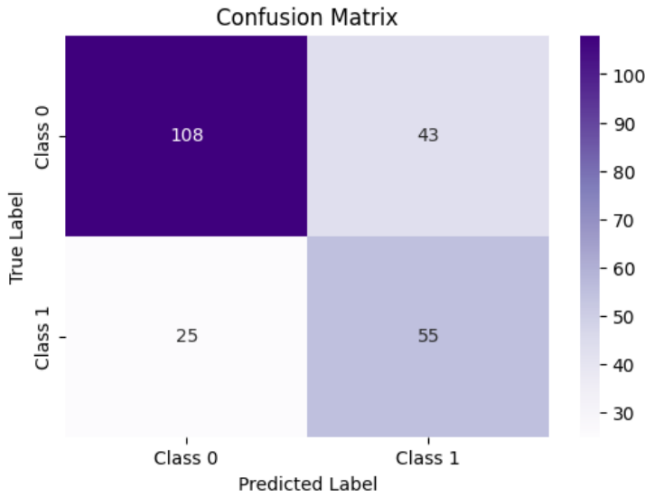


FIGURE 4.11: Confusion Matrix

The precision fig.(4.12) for class "0" is 81%, indicating that out of all the instances classified as "0," 81% of them are actually "0." The recall for class "0" is 72%, indicating that 72% of all "0" instances in the test set were correctly identified as such by the model.

For class "1," the precision is 56%, indicating that out of all the instances classified as "1," only 56% of them are actually "1." The recall for class "1" is 69%, which means that 69% of all "1" instances in the test set were correctly identified as such by the model.

The f1-score for class "0" is 0.76, while for class "1" it is 0.62. The accuracy weighted avg is 0.71, which takes into account the distribution of classes in the test set.

Classification Report:

	precision	recall	f1-score	support
0	0.81	0.72	0.76	151
1	0.56	0.69	0.62	80
accuracy			0.71	231
macro avg	0.69	0.70	0.69	231
weighted avg	0.73	0.71	0.71	231

FIGURE 4.12: Classification report

The area under the ROC curve (AUC) is 0.701, fig.(4.13) indicating a reasonable ability of the model to correctly classify positive versus negative instances.

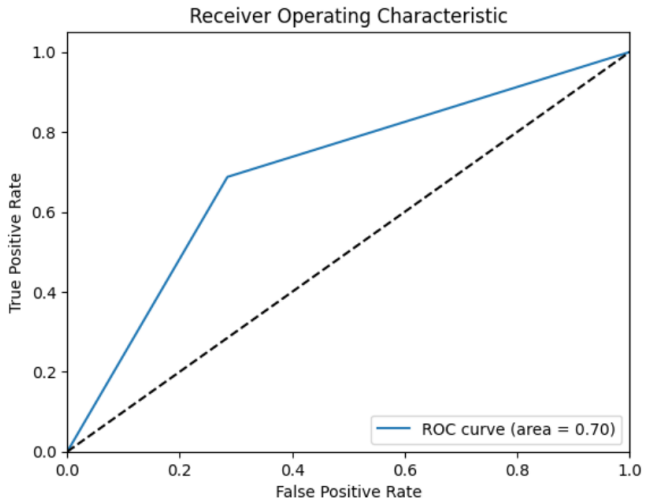


FIGURE 4.13: ROC

```
Accuracy scores for each fold: [0.75581395 0.68604651 0.75581395 0.73255814 0.85882353]
Mean accuracy: 0.7578112175102599
```

FIGURE 4.14: cross validation results

An average accuracy of 75.78% indicates that the SVM model with RBF kernel has a pretty good classification ability on the training set.
We conclude by saying that the SVM with RBF kernel optimized by Bayesian Optimization seems to provide similar results to the previous models.

Neural Network results

Precision measures the fraction of positive samples predicted correctly compared to all samples predicted as positive. In the neural network, the precision is 0.6037, which means that about 60% of the samples predicted as positive are actually positive.
The confidence interval is $[0.6977, 0.8088]$.

The confusion matrix fig.(4.15) shows the following results:

- 115 instances were correctly classified as "0" (true negatives).
- 36 instances were misclassified as "1" (false positives).
- 26 instances were misclassified as "0" (false negatives).
- 54 instances were correctly classified as "1" (true positives).

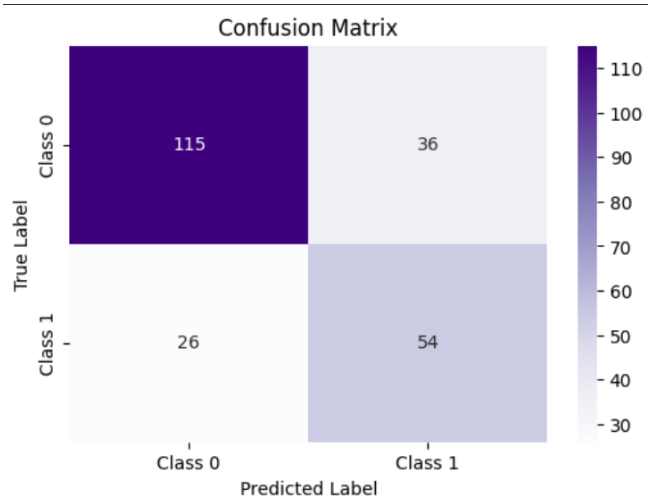


FIGURE 4.15: Confusion Matrix

The recall fig.(4.16) for class 0 is 0.76, which means that about 76% of the positive samples were correctly identified. The F1-score for class 0 is 0.79, and for class 1 is 0.64. accuracy is 0.73, which means that 73% of the samples were correctly classified. the weighted average of accuracy, recall and F1-score is 0.74.

Classification Report:

	precision	recall	f1-score	support
0	0.82	0.76	0.79	151
1	0.60	0.68	0.64	80
accuracy			0.73	231
macro avg	0.71	0.72	0.71	231
weighted avg	0.74	0.73	0.73	231

FIGURE 4.16: Classification Report

The area under the ROC curve (AUC) fig.(4.17) is an indication of the overall performance of the model. In the neural network, the AUC is 0.718, which suggests that the model has good discrimination ability between classes.

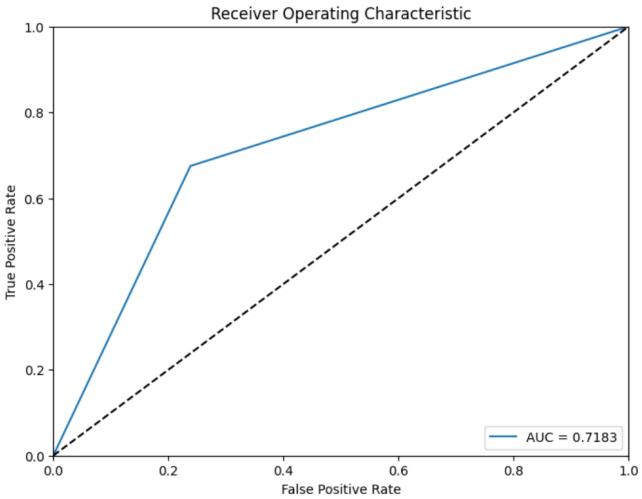


FIGURE 4.17: ROC

We conclude by saying that, the nn model seems to have reasonable performance in classifying non-diabetic samples (class 0), but it struggles a bit more in classifying diabetic samples (class 1), probably this is because of the imbalance in the data.

General Conclusion

In general, we can conclude by saying that all 4 models succeed in discretely classifying all the data in the dataset. We struggle more with instances of class 1, probably because the test set is more populated by instances of class 0.